Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#1: How to Install Custom BRK and /NMI Handlers

Revised by: Jim Mensch & Jim Merritt November 1988
Written by: Jim Merritt October 1986

This Technical Note discusses a method to install a custom debugger or debugging stub within the Apple IIGS system.

Introduction

This Technical Note discusses a particular method that you may use to install a custom debugger or debugging stub within the Apple IIGS system. The strategy and techniques described here should be of special interest to those who wish to operate the Apple IIGS as a slave to a debugger that resides on another machine.

Typically, an interrupt handler should pass control to a debugger or debugging stub whenever the processor executes a `BRK` instruction, or when an interface card triggers a non-maskable interrupt (/NMI). To simplify the design of the debugger, the Apple IIGS Monitor should be responsible for the following:

- saving all machine state information in locations that the debugger can access
- setting the machine to a known state
- passing control to an arbitrary debugger
- restoring the remembered machine state upon regaining control from the debugger
- resurrecting the interrupted process

The Monitor is designed to provide all of the services above for the `BRK` instruction, but only the third for /NMI interrupts. In addition, Apple II family systems are generally intolerant of /NMI interrupts. In this Technical Note we concentrate on the means by which you can install your own custom `BRK` handler, although we also briefly examine /NMI considerations.

Dealing With BRK

A `BRK` interrupt handler may reside at any address in memory. The Monitor passes control to your code by executing a `JSL` instruction; consequently, your routine must terminate with an `RTL` instruction. To install your `BRK` handler, simply load it into memory, call the Miscellaneous Tool Set `GetVector` routine to fetch the address of the current `BRK` handler, put

that address in a safe place, then supply the address of your handler to the Miscellaneous Tool Set `SetVector` routine. To deactivate your handler, restore the previous handler address using `SetVector` as follows:

```
;
;  NOTE: All Listings are in APW assembler format.
;

INSTMYBRK     anop                ;Example code to install user's BREAK handler.
              PushLong #0         ;Space for function call result.
              PushWord #$1C       ;We want BREAK vector address.
              _GetVector          ;Make the call using standard macro.

;  The stack now holds address of the current break handler.
              PLA                 ;Get and save low word of address…
              STA    SBRKADR
              PLA                 ; …and now high word.
              STA    SBRKADR+2
              PushWord #$1C       ;We want to change BREAK vector address.
              PushLong #MYHANDLR  ;Address of user's BRK handler.
              _SetVector          ;Make the call using standard macro.

;  Custom handler is in place, now go off and do whatever we like…

DEACMYBRK     anop                ;Example code to deactivate the BRK handler.
              PushWord #$1C       ;We want to change BREAK vector address.
              PushLong SBRKADR    ;The previous BRK handler address.
              _SetVector          ;Make the call using standard macro.
```

Upon entry to your code, the machine will be in eight-bit native mode. Specifically, the m and x bits will be set (forcing eight-bit accumulator, memory access, and index registers), the processor will be running at the normal (1 MHz) speed, all memory shadowing will be enabled, and both the direct page and data bank registers will be reset to zero. The same conditions must hold when your BRK handler returns control to the Monitor. While your code is active, however, it is free to affect the machine state in arbitrary ways, including (but not limited to) widening the registers, increasing the clock rate, and disabling shadowing. Before returning control to the Monitor, your break handler must also clear the processor's carry flag, as an indication that the BRK was indeed serviced by an external handler. (Note: The default BREAKVECTOR points to a "no-op" handler that simply sets the carry flag to indicate that there is no external handler available, and it then executes an RTL.)

When a BRK occurs, the processor saves the machine's state in the BRK.VAR area, and you may obtain this address with the Miscellaneous Tool Set GetAddr routine as follows:

```
              PushLong #0         ; space for result
              PushWord #9         ; we want BRK.VAR address
              _GetAddr            ; make the call using standard macro

; The stack now holds the address of the BRK.VAR area, expressed as a long word (four
bytes).
```

Coping With /NMI

Handling /NMI interrupts is, by far, a trickier proposition than fielding BRK instructions. For example, the user-definable /NMI jump-vector, /NMI ($0003FB), only has room in its three-byte JMP-absolute instruction for a two-byte address. Because of this size limitation, at least the "front end" of any /NMI handler must reside in bank $00. In addition, the Monitor does not "condition" the system in any way before transferring control through the /NMI hook, so the

system could be in native mode, emulation mode, or any hybrid mode (with any screen condition) upon entry to your handler. (Note: Although the 65816 processor provides for separate /NMI vector addresses in native and emulation modes, the Apple IIGS implementation of these two vectors pass control to the same user hook at $0003FB.) The processor only saves minimal machine state information when an /NMI occurs; if the handler needs to preserve more than the program counter and status register (which are saved automatically), then it must do so explicitly. Because the 65816 assumes any program running in emulation mode has its program bank register in bank zero, it will not save the program bank register for any program running in emulation mode outside of bank zero. Code which runs in this manner will always crash if it makes any attempt to return from the interrupt. Finally, /NMI interrupts can create havoc with disk access and other aspects of the system; consequently, the only way you can safely use /NMI interrupts is as a one-way "escape hatch" to emergency debugging code.

Here are some ground rules for /NMI interrupt handlers.

- On entry, store any interesting registers or machine state in RAM space owned by the handler.
- Determine whether the processor is in emulation mode or native mode.
- Take appropriate action, depending upon the processor mode.
- Under no circumstances try to return from the interrupt! Restart the system instead.

To install an /NMI handler, load it into some free RAM in bank $00, put the two-byte address currently at location /NMI+1 in a safe place, then replace it with the address of your handler. To deactivate your handler (assuming nothing has yet invoked it), simply restore the previous handler address to /NMI+1.